Continuous Integration der Diagnosetoolkette der Daimler AG

**Bachelorarbeit**

im Studiengang  
Medieninformatik

vorgelegt von

**Malte Leon Lohrer**Matr.-Nr.: 27399

am 05. April 2018   
an der Hochschule der Medien Stuttgart

Erstprüfer/in: Prof. Dr. Ansgar Gerlicher  
Zweitprüfer/in: Sebastian Gerber

# Ehrenwörtliche Erklärung

„Hiermit versichere ich, Malte Lohrer ehrenwörtlich, dass ich die vorliegende Bachelorarbeit (bzw. Masterarbeit) mit dem Titel: „Continuous Integration der Diagnosetoolkette der Daimler AG“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.“

01.04.2018

# Kurzfassung

# Abstract

# Inhaltsverzeichnis

Ehrenwörtliche Erklärung 2

Kurzfassung 3

Abstract 3

Inhaltsverzeichnis 4

Abbildungsverzeichnis 5

Tabellenverzeichnis 5

Abkürzungsverzeichnis 6

Vorwort 7

1 Überblick 8

2 Ziele 9

3 Stand der Technik 11

4 Continuous Integration 12

4.1 Das Konzept der Continuous Integration 12

4.1.1 Gemeinsame Codebasis 13

4.1.2 Automatisierter Build 13

4.1.3 Selbsttestender Build 13

4.1.4 Häufige Integration 14

4.1.5 Builds und Tests nach jeder Änderung 14

4.1.6 Schnelle Build-Zyklen 15

4.1.7 Tests in gespiegelter Produktionsumgebung 15

4.1.8 Einfacher Zugriff auf Build-Ergebnisse 15

4.1.9 Automatisierte Berichte 15

4.1.10 Automatisierte Verteilung 15

Quellenverzeichnis 16

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

CI Continuous Integration

GUI Graphical User Interface (Deutsch: Grafische Benutzeroberfläche)

HdM Hochschule der Medien

# Vorwort

# Überblick

# Ziele

Ziel der vorliegenden Arbeit ist es, das Konzept der *Continuous Integration* genau zu beleuchten und theoretischauf die Diagnosetoolkette der Daimler AG anzuwenden. Die dabei gewonnenen Erkenntnisse werden die Vor- und Nachteile einer CI aufzeigen.

Der praktische Teil der Thesis, welcher auf der beiliegenden CD enthalten ist, stellt einen ersten Schritt in Richtung CI der Diagnosetoolkette dar. Ziel dieses Teils ist es, das Diagnosetool *Monaco* automatisiert zu testen um damit die Funktionalität des Tools zuverlässig gewährleisten zu können.

Nachfolgend sind die Feinziele der Praktischen Arbeit aufgelistet:

1. Gemeinsame Codebasis (erfüllt)
   1. Gerade im Hinblick auf eine CI ist es enorm wichtig direkt von Anfang an die Möglichkeit einer gemeinsamen Codebasis zu schaffen. Diese wird durch ein *GitLab Repository* realisiert, welches in Zukunft problemlos von anderen Teams genutzt werden kann.
2. Modularer Code (erfüllt)
   1. Ein weiteres Ziel ist es den Quellcode in möglichst kleine Module aufzuteilen. Das bringt nicht nur durch eine bessere Übersicht Vorteile, sondern kommt auch der Wartbarkeit des Quellcodes enorm entgegen. Außerdem lassen sich kleinere Teile Quellcode deutlich leichter, effizienter und vor allem abgekapselt testen, was bei einer guten CI eine große Rolle spielt.
3. Automatisierung der grafischen Benutzeroberfläche von *Monaco (erfüllt)*
   1. Ein wesentliches Ziel besteht darin, das Tool *Monaco*, auf welche in Kapitel <X> genauer eingegangen wird, zu automatisieren. Da diese Anwendung über eine Grafische Benutzeroberfläche (im Folgenden als „GUI“ bezeichnet) gesteuert wird, müssen hierbei die Aktionen die normalerweise ein Mensch durchführt, vom Code übernommen werden.
4. Bereitstellen von Unit Tests (erfüllt)
   1. Unverzichtbar für eine richtig durchgeführte CI ist es, genügend Tests zu implementieren, welche stets die Funktion der einzelnen Codebestandteile sicherstellen. Hierbei werden sogenannte *Unit Tests* durchgeführt, welche jeder für sich nur sehr kleine des Quellcodes testen, dafür jedoch sehr schnell und präzise.
5. Bereitstellen von Komponenten- und Systemtests (nicht erfüllt)
   1. Anders als bei Unit Tests werden bei Komponententests zusammenhängende Codebereiche getestet. Das sind dann zum Beispiel mehrere Klassen, welche im gesamten Programmablauf zusammenarbeiten.
   2. Eine noch umfassendere Art von Tests stellen Systemtests dar, welche Grundsätzlich das gesamte Produkt testen. Hierbei werden tatsächlich alle Teile der Software im Zusammenspiel getestet. Darunter fallen beispielsweise auch GUI-Tests.
6. Automatisches generieren der Ergebnisse (erfüllt)
   1. Die Nachvollziehbarkeit des Programmdurchlaufs stellt ein weiteres Ziel dar. Diese wird durch das protokollieren der Ergebnisse der einzelnen Teilabschnitte eines jeden Durchlaufs gewährleistet. Nach einem vollständigen Programmablauf werden Daten bereitgestellt, welche genau aufzeigen welche Funktionen zu welchem Zeitpunkt abgelaufen sind und ob es dabei zu Fehlern kam.
7. Abhängigkeiten zwischen den Funktionen klar definieren (erfüllt)
   1. Es muss gewährleistet sein, dass bestimmte Teile des Codes erst dann ausgeführt werden, wenn andere Teile schon erfolgreich durchlaufen wurden. Diese Abhängigkeiten müssen klar definiert werden. Wenn eine Funktion fehlschlägt, werden nachfolgende erst gar nicht ausgeführt. Diese Abhängigkeiten wurden mithilfe von *luigi* realisiert, ein Python Modul auf welches in Kapitel <X> genauer eingegangen wird.
8. Visualisierung des gesamten Programmablaufs (erfüllt)
   1. Eines der bedeutendsten Ziele stellt die Visualisierung des Programmablaufs dar. Wie schon bei der Definition der Abhängigkeiten, wird hier das Python Modul *luigi* genutzt, um stets den Status des Ablaufs in grafisch aufbereiteter Form auf einer Website nachvollziehen zu können.

# Stand der Technik

Hier werde ich die ganze Toolkette von Daimler beschreiben. Wie Läuft Sie ab welche Tools werden benutzt wie genau wird hier gearbeitet – vor allem natürlich in Hinblick auf CI .

Die Toolkette werde ich vermutlich in 3.1, 3.2 , 3.3 usw. unterteilen

# Continuous Integration

Dieses Kapitel befasst sich mit der Continuous Integration als Solche und gibt dem Leser einen Einblick in die Softwareentwicklung unter Anwendung von CI.

Hierfür werden die Grundprinzipien einer CI erläutert und es wird auf deren Vor- und Nachteile eingegangen.

## Das Konzept der Continuous Integration

Die Continuous Integration ist nach Wiest (2010, S.13) eine Softwareentwicklungspraktik, welche erstmals als eine der Praktiken der sogenannten Extremprogrammierung (*eXtreme Programming*) populär wurde.

Der Begriff *Continuous Integration* entspringt dem gleichnamigen Artikel von Martin Fowler, welcher das erste Mal im Jahr 2000 erschien und im Jahr 2006 noch einmal überarbeitet und aktualisiert wurde.

In eben diesem Artikel beschreibt Fowler (2006, Erster Abschnitt) das Grundkonzept der CI folgendermaßen:

“Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. This article is a quick overview of Continuous Integration summarizing the technique and its current usage.”

Grundsätzlich geht es bei CI nach Fowler darum, dass die Softwareentwickler die Änderungen welche sie am Code vornehmen so oft wie möglich integrieren, um die Deltas zwischen der bisherigen und der neuen Version möglichst gering zu halten. Der neue Softwarestand wird dann umgehend gebaut, wodurch Fehler, durch die kleinere Menge an Änderungen im Vergleich zum letzten Softwarestand, viel schneller gefunden werden können. Dadurch soll der Integrationsvorgang selbst zu einem *Non-Event* (dt.: nicht-Ereignis) werden (vgl. Martin Fowler, 2006).

Fowler formulierte in seinem Artikel „Continuous Integration“ außerdem insgesamt 10 Praktiken, welche für eine effektive CI maßgeblich sind.

Für die Namen dieser Praktiken werden im Folgenden die deutschen Übersetzungen von Dr. Simon Wiest (2010, S. 15-16) verwendet.

### Gemeinsame Codebasis

Sämtliche Daten, welche zu einem Softwareprojekt gehören, müssen in einem für alle an dem Projekt beteiligten Akteure offen und jederzeit zugänglich sein. Meist ist hier die Rede von einem Versionskontrollsystem.

Wichtig hierbei ist, dass dort tatsächlich *alle Daten* abgelegt werden, die zum Bauen eines Builds nötig sind, abgelegt werden.

### Automatisierter Build

Das Projekt muss stets aus der Summe seiner Teile vollautomatisch und von Grund auf neu gebaut werden können. Fowler (2006) beschreibt hier zusätzlich, dass es Sinn macht Build-Werkzeuge zu nutzen, welche auf allen nötigen Plattformen zur Verfügung stehen, da Build-Werkzeuge von Entwicklungsumgebungen oftmals proprietäre Dateien beinhalten und damit nicht zwingend auf jedem System funktionieren werden.

### Selbsttestender Build

Während des Build-Prozesses muss das Produkt automatisch getestet werden. Es obliegt dem Entwickler diese Tests sinnvoll zu implementieren. Es ist wünschenswert, so viel Quellcode wie möglich durch Tests abzudecken. Dabei unterscheidet man zwischen

sehr feingranularen Tests wie z.B. den *Unit-Tests*, welche nur sehr kleine, meist unabhängige Abschnitte Testen.

den mittelgroßen *Komponententests* die, wie der Name verrät, ganze Komponenten testen – also beispielsweise die Funktion mehrerer miteinander kommunizierender Klassen.

Den übergreifenden *Systemtests*, welche ein Produkt in seiner Ganzheit testen.

### Häufige Integration

Einer der offensichtlichsten und wichtigsten Punkte dieser Auflistung ist die häufige Integration von Code. Die Entwickler sind dazu angehalten, so oft wie möglich, mindestens jedoch 1 Mal am Tag, Ihren Code in ein Versionskontrollsystem einzuchecken. Simon Wiest (2006, S.34) spricht hier von einer kulturellen Veränderung, weil das häufige Einchecken des Quellcodes etwas ist, dass sich viele Entwickler erst angewöhnen müssen, bis es selbstverständlich wird. Bevor eingecheckt werden darf, muss natürlich die Version vorab lokal erfolgreich gebaut werden können um auf diese Weise zu vermeiden, dass Fehlerhafter Quellcode eingecheckt wird. Die Vorteile der häufigen Integration liegen auf der Hand:

Der aktuelle Entwicklungsstand und das aktuelle Master Build liegen nie weit auseinander, was enorm bei der Fehlerfindung hilft, da neue Fehler nur in dem neuen Delta zwischen den beiden Versionen auftreten verursacht werden können.

Die Entwickler können Risikofreudiger entwickeln, da sie jederzeit einen sogenannten *Roll-back* auf den letzten aktuellen Stand durchführen können.

Es kann viel schneller auf Fehler reagiert werden.

Man hat zu jedem Zeitpunkt eines Projekts einen funktionierenden Stand, auch wenn das nicht heißt, dass dieser fertig ist.

### Builds und Tests nach jeder Änderung

Ein absolut essenzieller Punkt einer CI ist, ebenso wie das häufige integrieren von neuem Code, das ebenso häufige builden und testen des selbigen. Neuer Quellcode sollte bei einer guten CI in der Theorie nach jeder Änderung getestet und anschließend, bei erfolgreichen Tests, gebaut werden. Zu häufiges testen und builden kann nach Wiest (2010, S. 38) aber zu einem Rückstau an Builds führen, bzw. zu Zeitintensiv ausfallen. Dr. Simon Wiest schlägt hier als groben Lösungsansatz vor, so oft wie es die Gegebenheiten zulassen zu builden und nicht Zwingend nach jeder Änderung. Wie oft man den Quellcode tatsächlich kompiliert und baut, hängt letztendlich von individuellen Faktoren ab und muss an diese angepasst werden.

Martin Fowler (2006) beschreibt, dass es Sinnvoll ist einen CI Server zu verwenden, welcher den Quellcode Verwaltet bzw. überwacht. Wenn ein Projektmitglied einen neuen Stand zu dem verwalteten Repository[[1]](#footnote-1) hinzufügt, bzw. einen sogenannten *commit*[[2]](#footnote-2)ausführt, dann stößt der CI Server den Build-Vorgang an und benachrichtigt das Mitglied über das Ergebnis. Durch die Verwendung eines CI Servers wird sichergestellt, dass nach *jedem* Commit ein Build erzeugt wird und die Entwickler zeitnah die Ergebnisse erhalten. Ein CI Server setzt allerdings wiederum schnelle Build-Zyklen voraus.

### Schnelle Build-Zyklen

TEST

### Tests in gespiegelter Produktionsumgebung

### Einfacher Zugriff auf Build-Ergebnisse

### Automatisierte Berichte

### Automatisierte Verteilung

# Quellenverzeichnis

**Dr. Simon Wiest** (2010): Continuous Integration mit Hudson, Grundladen und Praxiswissen für Einsteiger und Umsteiger. Heidelberg: dpunkt.Verlag

**Martin Fowler** (2006): Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html (23. Februar 2018)

**Wikipedia, Die freie Enzyklopädie (**21. Juni 2017): Seite „Repository“.<https://de.wikipedia.org/w/index.php?title=Repository&oldid=166597004> (26. Februar 2018)

**Wikipedia, Die freie Enzyklopädie (**18. August 2015): Seite „Commit“.<https://de.wikipedia.org/w/index.php?title=Commit&oldid=145145351> (26. Februar 2018)

1. Ein Repository ist ein verwaltetes Verzeichnis zur Speicherung und Beschreibung von digitalen Objekten für ein digitales Archiv (Wikipedia 21. Juni 2017, Seite „Repository“) [↑](#footnote-ref-1)
2. Commit ist ein Ausdruck aus der [Softwaretechnik](https://de.wikipedia.org/wiki/Softwaretechnik), der die bestätigende [Freischaltung](https://de.wikipedia.org/wiki/Freischaltung) einer Änderung beschreibt (Wikipedia 18. August 2015, Seite „Commit“) [↑](#footnote-ref-2)